

WolfSSL

Report for the Information System Security exam at the Politecnico di
Torino

Luca Valentini (278487)

september 2020

Contents

1	SSL Protocol	1
1.1	Introduction	1
1.2	SSL Handshake	3
2	WolfSSL	5
2.1	Introduction	5
2.2	Operating system supported	6
2.3	WolfSSL's features	7
3	Test case	8
3.1	Client/Server provided by WolfSSL	8
4	Create a program using WolfSSL	10
4.1	TCP application	10
4.2	From TCP to TLS	11
4.3	TLS application	12
4.3.1	iterative program	12
4.4	SSL Server	13
4.5	TLS Client	19
4.6	Compile a WolfSSL program	23
4.7	Execute a WolfSSL program	23
5	Differences between WolfSSL and OpenSSL	29
5.1	Differences	29
5.2	Build size and speed test	30

Abstract

This document was intended as a small introductory guide to wolfssl.

For the creation of this thesis I used the official wolfssl manual and some github repository related to wolfssl.

1 SSL Protocol

1.1 Introduction

The SSL protocol is a client/server protocol that provides the following basic security services to the communicating peers:

- Authentication (both peer entity and data origin authentication) services
- Connection confidentiality services
- Connection integrity services

The SSL protocol is sockets-oriented, meaning that all or none of the data that is sent to or received from a network connection is cryptographically protected in exactly the same way. It can be best viewed as an intermediate layer between the transport and the application layer that serves two purposes:

- Establish a secure connection between the communicating peers
- Use this connection to securely transmit higher-layer protocol data from the sender to the receiver. It therefore fragments the data in pieces called fragments; each fragment is optionally compressed, authenticated, encrypted, prepended with a header, and transmitted to the receiver. Each data fragment prepared this way is sent in a distinct SSL record.

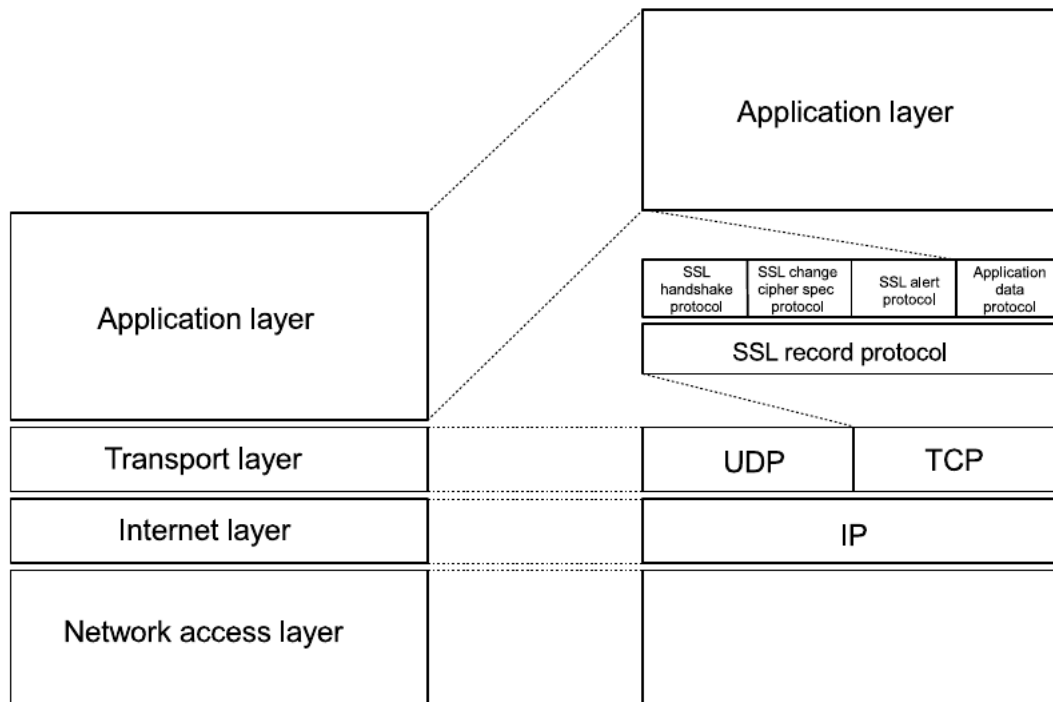


Figure 1: The SSL with its (sub)layer and (sub)protocols

The SSL consists of two sublayers and a few subprotocols:

- The lower sublayer is stacked on top of some connection-oriented and reliable transport layer protocol. This layer basically comprises the SSL record protocol that is used for the encapsulation of the higher-layer protocol data.
- The higher sublayer is stacked on top of the SSL record protocol and comprises four subprotocols.
 - The *SSL handshake protocol* is the core subprotocol of SSL. It is used for establishment of a secure connection. It allows the communicating peers to authenticate each other and to negotiate a cipher suite and a compression method.
 - The *SSL change cipher spec protocol* is used to put the parameters, set by the SSL handshake protocol in place and make them effective.
 - The *SSL alert protocol* allows the communicating peers to signal indicators of potential problems and send respective alert messages to each other.

- The *SSL application data protocol* is used for the secure transmission of application data.

In spite of the fact that SSL consists of several subprotocols, we use the term *SSL protocol* to refer to all of them simultaneously.

1.2 SSL Handshake

The SSL handshake protocol is layered on top of the SSL record protocol. It allows a client and server to authenticate each other and to negotiate issues like cipher suites and compression methods.

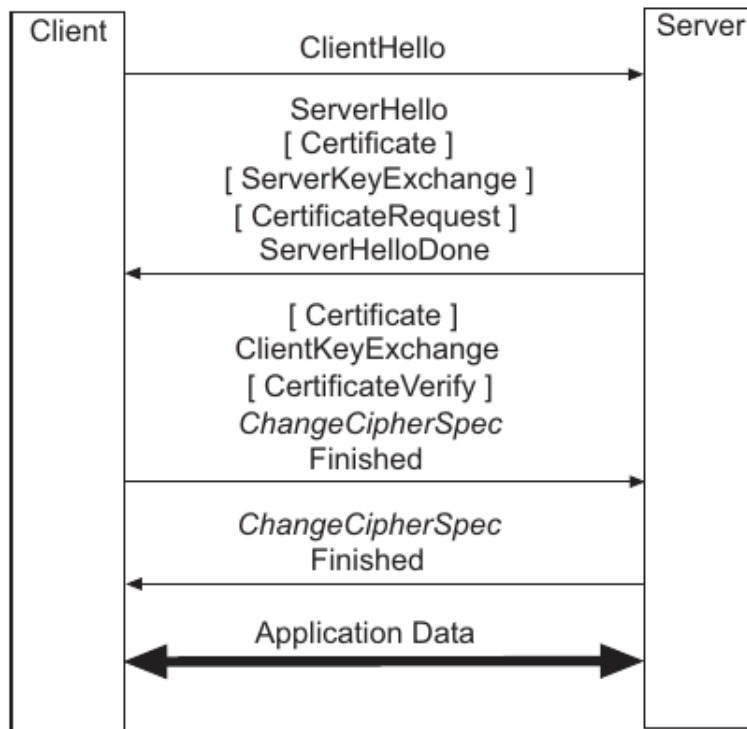


Figure 2: The SSL handshake protocol

The SSL handshake protocol comprises four sets of messages:

- The first flight comprises a single *ClientHello* message that is sent from the client to the server.
- The second flight comprises two messages that are sent back from the server to the client:

1. *ServerHello* message is sent in response to the *ClientHello* message
 2. (optional) If the server is to authenticate itself, it may send a *Certificate* message to the client.
 3. (optional) Under some circumstances, the server may send a *ServerKeyExchange* message to the client.
 4. (optional) If the server requires the client to authenticate itself with a public key certificate, then it may send a *CertificateRequest* message to the client.
 5. Finally, server send a *ServerHelloDone* message to the client.
- The third flight comprises three to five messages that are again sent from the client to the server:
 1. (optional) If the server has sent a *CertificateRequest* message, then the client sends a *Certificate* message to the server.
 2. In the main step of the protocol, the client sends a *ClientKeyExchange* message to the server.
 3. (optional) If the client has sent a certificate to the server, then it must also send a *CertificateVerify* message to the server. This message is digitally signed with the private key that corresponds to the client certificate's public key.
 4. The client sends a *ChangeCipherSpec* message to the server (using the SSL change cipher spec protocol) and copies its pending write state into the current write state.
 5. The client sends a *Finished* message to the server. As mentioned above, this is the first message that is cryptographically protected under the new cipher spec.
 - The fourth flight comprises two messages that are sent from the server back to the client:
 1. The server sends another *ChangeCipherSpec* message to the client and copies its pending write state into the current write state.
 2. Finally, the server send a *Finished* message to the client. Again, this message is cryptographically protected under the new cipher spec.

At this point in time, the SSL handshake is complete and the client and server may start exchanging application-layer data.

2 WolfSSL

2.1 Introduction

The wolfSSL embedded library is a lightweight TLS library written in ANSI C and targeted for embedded, RTOS, and resource-constrained environments - primarily because of its small size, speed, and feature set; It's an SSL/TLS library optimized to run on embedded platforms. It's free and it has an excellent cross platform support.

WolfSSL supports SSL 3.0, TLS(1.0, 1.1, 1.2, 1.3), and DTLS(1.0, 1.2). It also includes an OpenSSL compatibility interface with the most commonly used OpenSSL functions. WolfSSL is open source, licensed under the GNU General Public License GPLv2.

This library is built for maximum portability and supports the C programming language as a primary interface. It also supports several other host languages, including Java (wolfSSL JNI), C# (wolfSSL C#), Python, and PHP and Perl.

To improve performance it supports hardware cryptography and acceleration on several platforms.

WolfSSL uses the following cryptography libraries:

- wolfCrypt
 - Provides RSA, ECC, DSS, Diffie-Hellman, EDH, NTRU, DES, Triple DES, AES (CBC, CTR, CCM, GCM), Camellia, IDEA, ARC4, HC-128, ChaCha20, MD2, MD4, MD5, SHA-1, SHA-2, SHA-3, BLAKE2, RIPEMD-160, Poly1305, Random Number Generation, Large Integer support, and base 16/64 encoding/decoding.
- NTRU
 - An open source public-key cryptosystem that uses lattice-based cryptography to encrypt and decrypt data.

2.2 Operating system supported

The operating systems supported are:

- | | | |
|--------------------|--|---------------------------|
| 1. Win32/64 | 18. Nintendo Wii and Gamecube through DevKit-Pro | 31. ARC MQX |
| 2. Linux | | 32. TI - RTOS |
| 3. Mac OS X | | 33. uTasker |
| 4. Solaris | 19. QNX | 34. embOS |
| 5. ThreadX | 20. MontaVista | 35. INtime |
| 6. VxWorks | 21. NonStop | 36. Mbed |
| 7. FreeBSD | 22. TRON / ITRON / ITRON | 37. uT - Kernel |
| 8. NetBSD | 23. Micrium C / OS - III | 38. RIOT |
| 9. OpenBSD | | 39. CMSIS -RTOS |
| 10. embedded Linux | 24. FreeRTOS | 40. FROSTED |
| 11. Yocto Linux | 25. SafeRTOS | 41. Green Hills INTEGRITY |
| 12. OpenEmbedded | 26. NXP / Freescale MQX | 42. Keil RTX |
| 13. WinCE | | 43. TOPPERS |
| 14. Haiku | 27. Nucleus | 44. PetaLinux |
| 15. OpenWRT | 28. TinyOS | 45. Apache Mynewt |
| 16. iPhone(iOS) | 29. HP / UX | 46. PikeOS |
| 17. Android | 30. AIX | |

2.3 WolfSSL's features

- Runtime memory usage between 1-36 kB
- Minimum footprint size of 20-100 kB, depending on build options and operating environment
- OpenSSL compatibility layer
- Hash Functions:
 - MD2
 - MD4
 - MD5
 - SHA-1
 - SHA-224
 - SHA-256
 - SHA-384
 - SHA-512
 - BLAKE2b
 - RIPEMD-160
 - Poly1305
- OCSP, OCSP Stapling, and CRL support
- Block, Stream, and Authenticated Ciphers:
 - AES (CBC, CTR, GCM, CCM, GMAC, CMAC), Camellia, DES, 3DES, IDEA, ARC4, RABBIT, HC-128, ChaCha20
- Public Key Algorithms:
 - RSA, DSS, DH, EDH, ECDH-ECDSA, ECDHE-ECDSA, ECDH-RSA, ECDHE-RSA, NTRU
- Password-based Key Derivation: HMAC, PBKDF2
- Curve25519 and Ed25519
- ECC and RSA Key Generation
- X.509v3 RSA and ECC Signed Certificate Generation
- PEM and DER certificate support
- Modular cryptography library (wolfCrypt)
- Open Source Project Integrations:
 - MySQL, OpenSSH, Apache httpd, Open vSwitch, stunnel, Lighttpd, GoAhead, Mongoose, and more!
- PKCS#1 (RSA Cryptography Standard) support

- PKCS#3 (Diffie-Hellman Key Agreement Standard) support
- PKCS#5 (Password-Based Encryption Standard) support
- PKCS#7 (Cryptographic Message Syntax - CMS) support
- PKCS#8 (Private-Key Information Syntax Standard) support
- PKCS#9 (Selected Attribute Types) support
- PKCS#10 (Certificate Signing Request - CSR) support
- PKCS#11 (Cryptographic Token Interface) support
- PKCS#12 (Certificate/Personal Information Exchange Syntax Standard) support
- Mutual authentication support (client/server)
- SSL Sniffer (SSL Inspection) Support
- IPv4 and IPv6 support

3 Test case

3.1 Client/Server provided by WolSSL

```
server@server:~/Documents/information_system_security/wolfSSL/wolfssl/examples/server$ ./server -b
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL curve name is SECP256R1
Client message: hello wolfssl!
server@server:~/Documents/information_system_security/wolfSSL/wolfssl/examples/server$
```

Figure 3: Server TLS

+

```
luca@luca:~/Documents/information_system_security/wolfSSL/wolfssl/examples/client$ ./client -h 192.168.0.53
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL curve name is SECP256R1
I hear you fa shizzle!
luca@luca:~/Documents/information_system_security/wolfSSL/wolfssl/examples/client$
```

Figure 4: Client TLS

In this example, the server is a simple TLS server that allows only one client connection; after the connection with a client, the server receives an encrypted message from client, it responds and quits. The -b parameter allows the server to bind to any interface instead of localhost only.

After the connection with the server, the client sends a message (hello wolfssl!) and, after the server response, the client quits. The -h parameter allows the client to specify the server address to perform the connection.

ssl && ip.addr == 192.168.0.53						
No.	Time	Source	Destination	Protocol	Length	Info
121	12.460236481	192.168.0.56	192.168.0.53	TLSv1.2	234	Client Hello
123	12.462016137	192.168.0.53	192.168.0.56	TLSv1.2	161	Server Hello
125	12.466739263	192.168.0.53	192.168.0.56	TLSv1.2	2627	Certificate
127	12.552291703	192.168.0.53	192.168.0.56	TLSv1.2	404	Server Key Exchange
129	12.552368545	192.168.0.53	192.168.0.56	TLSv1.2	101	Certificate Request
131	12.552386638	192.168.0.53	192.168.0.56	TLSv1.2	75	Server Hello Done
133	12.552917310	192.168.0.56	192.168.0.53	TLSv1.2	1311	Certificate
135	12.562653252	192.168.0.56	192.168.0.53	TLSv1.2	141	Client Key Exchange
137	12.577135991	192.168.0.56	192.168.0.53	TLSv1.2	335	Certificate Verify
138	12.577489995	192.168.0.56	192.168.0.53	TLSv1.2	117	Change Cipher Spec, Encrypted Handshake Message
141	12.581720908	192.168.0.53	192.168.0.56	TLSv1.2	72	Change Cipher Spec
143	12.581790743	192.168.0.53	192.168.0.56	TLSv1.2	111	Encrypted Handshake Message
145	12.582040250	192.168.0.56	192.168.0.53	TLSv1.2	109	Application Data
147	12.583925279	192.168.0.53	192.168.0.56	TLSv1.2	117	Application Data
149	12.583979619	192.168.0.53	192.168.0.56	TLSv1.2	97	Encrypted Alert
152	12.584137112	192.168.0.56	192.168.0.53	TLSv1.2	97	Encrypted Alert

<ul style="list-style-type: none"> Frame 121: 234 bytes on wire (1872 bits), 234 bytes captured (1872 bits) on interface wlp60s0, id 0 Ethernet II, Src: IntelCor_f3:50:e8 (28:c6:3f:f3:50:e8), Dst: Raspberr_95:30:37 (dc:a6:32:95:30:37) Internet Protocol Version 4, Src: 192.168.0.56, Dst: 192.168.0.53 Transmission Control Protocol, Src Port: 55010, Dst Port: 11111, Seq: 1, Ack: 1, Len: 168 Transport Layer Security
--

Figure 5: All TLS packets sent

Client IP: 192.168.0.47

Server IP: 192.168.0.53

As you can see from the figure above, the communication is initialized from client with a 'Client Hello'; after TLS handshake, there are 2 messages 'Application Data' sent respectively from the client to the server and from the server to the client, whose content is encrypted. Once the server sends a response to the client, the communication is closed.

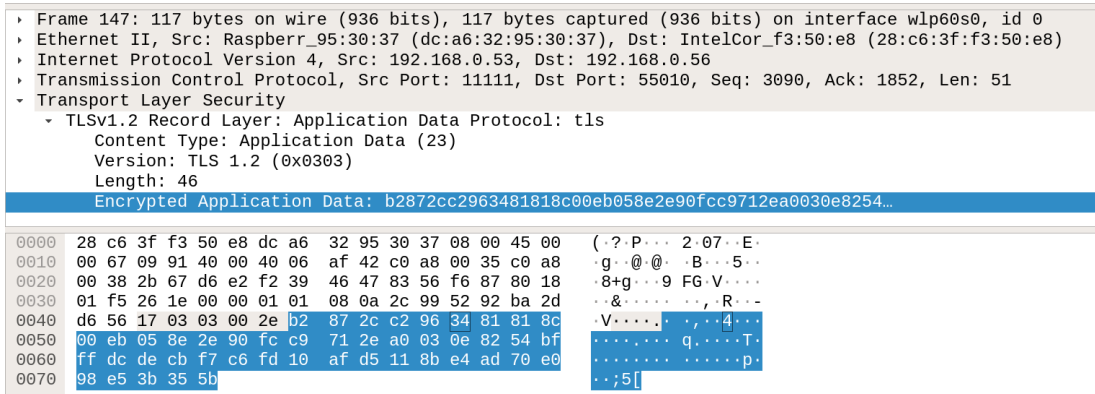


Figure 6: Content of the encrypted message

4 Create a program using WolfSSL

4.1 TCP application

To create a TLS program you can modify your TCP program by adding several TLS functions. To explain the migration from TCP to TLS, I created a simple chat between a client and a server. The code of this program isn't reported in this text but it is available on [github](#).

In the picture there is a wireshark screenshot to see the data traffic of a tcp application; this image with the next sections allows to understand the difference between an application with or without encryption.

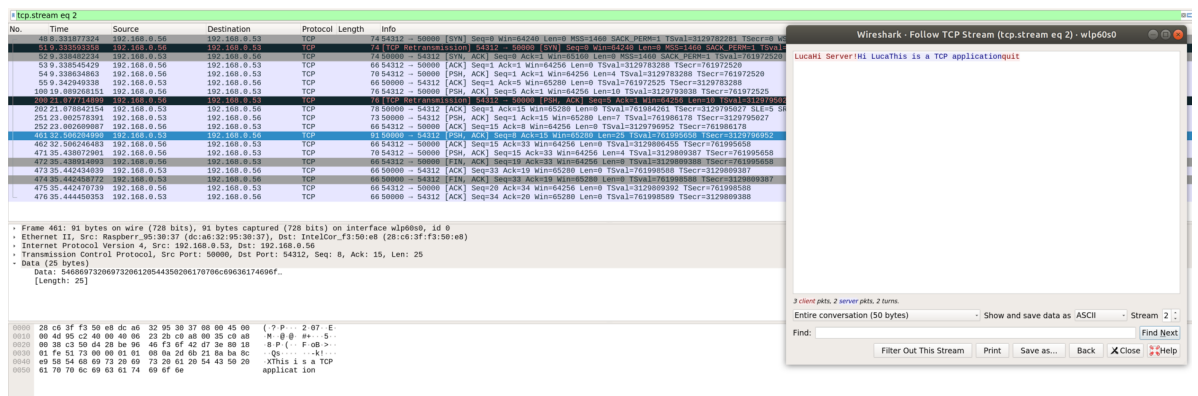


Figure 7: TCP data traffic

4.2 From TCP to TLS

To create a wolfSSL application the first thing that I did was including the wolfSSL API header in my program.

```
#include <wolfssl/ssl.h>
```

After the inclusion of the header files, I initialized the library and the WOLFSSL_CTX calling **wolfSSL_Init**; it is necessary to use the library.

The WOLFSSL_CTX structure contains global values for each SSL connection, including certificate information. To create a new WOLFSSL_CTX there is **wolfSSL_CTX_new()** function. It requires an argument which defines the SSL or TLS protocol for the client or server to use. In my case I used TLS 1.3, so the call is:

```
WOLFSSL_CTX *ctx = wolfSSL_CTX_new(wolfTLSv1_3_server_method());
```

for the server;

```
WOLFSSL_CTX *ctx = wolfSSL_CTX_new(wolfTLSv1_3_client_method())
```

for the client.

In the WOLFSSL_CTX the CA (Certificate Authority) can be loaded so that the client is able to verify the server's identity when they start the connection. To load the CA into the WOLFSSL_CTX there is **wolfSSL_CTX_load_verify_locations()**. This function requires three arguments:

- a WOLFSSL_CTX pointer
- a certificate file
- a path value that point to a directory which should contain CA certificates in PEM format.

this function returns SSL_SUCCESS or SSL_FAILURE.

wolfSSL_CTX_load_verify_locations() can be used to verify the certificate of the servers by the client. The server loads a certificate file into the TLS context (WOLFSSL_CTX) through this function:

```
int wolfSSL_CTX_use_certificate_file(ctx, CERT_FILE,  
    SSL_FILETYPE_PEM);
```

Then the server must load the private key with:

```
int wolfSSL_CTX_use_PrivateKey_file(ctx, KEY_FILE,
    SSL_FILETYPE_PEM)
```

After a TCP connection the WOLFSSL object needs to be created and the file descriptor needs to be associated with the session; the instructions are:

```
//Connect to socket file descriptor
WOLFSSL* ssl;
//create WOLFSSL object
ssl = wolfSSL_new(ctx);
wolfSSL_set_fd(ssl,sockfd);
```

After the previous instructions called by client and server, the server waits for a TLS client to initiate the SSL handshake; it waits until a client calls **wolfSSL_connect(ssl)** and then the handshake starts.

Once the connection functions were set, I replaced **read(...)** function with:

```
int wolfSSL_read(WOLFSSL *ssl, void *data, int sz);
```

It read **sz** bytes from the SSL session **ssl** internal read buffer into the buffer **data**. The bytes are removed from the internal receive buffer;

Instead the **write(...)** function is replaced by:

```
int wolfSSL_write(WOLFSSL *ssl, void *data, int sz);
```

It writes **sz** bytes from the buffer, **data**, to the SSL connection, **ssl**.

When the application is over, the WOLFSSL_CTX object and the wolfSSL library must be freed; the instructions are:

```
wolfSSL_free(ssl);
wolfSSL_CTX_free(ctx);
wolfSSL_Cleanup();
```

4.3 TLS application

4.3.1 iterative program

Explanation of the iterative program developed by me; remember to describe how it work, especially say that is a 1-1 client server; client can write only if the server write before.

To develop a TLS application, I started from the TCP chat explained in the previous subsection and I added the wolfSSL function to create an encrypted communication.

The goal of this program is to show how to create a simple encrypted chat, focusing on the security of an application, rather than on the good practices of the socket and thread applications.

Some parts of the code like inclusion and global variables are omitted (wolfSSL object and other variables are stored in global memory);

In the next sections I'll try to explain the code of the program:

4.4 SSL Server

For testing purpose, all the certificates that I use and the server private key were taken from wolfSSL download.

After the initialization of the wolfSSL and the socket, the main function marks the socket referred to by sockfd as a passive socket, that is, as a socket that will be used to accept incoming connection requests using accept.

```
int main()
{
    int ret;

    /* Initialize wolfSSL */
    wolfSSL_Init();

    /* Create a socket that uses an internet IPv4 address,
     * Sets the socket to be stream based (TCP),
     * 0 means choose the default protocol. */
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        fprintf(stderr, "ERROR: failed to create the socket\n");
        return -1;
    }
    /* Create and initialize WOLFSSL_CTX */

    if ((ctx = wolfSSL_CTX_new(wolfTLSv1_3_server_method())) ==
        NULL)
    {
        fprintf(stderr, "ERROR: failed to create WOLFSSL_CTX\n");
        return -1;
    }

    /* Load server certificates into WOLFSSL_CTX */
```



```

if (wolfSSL_CTX_use_certificate_file(ctx, CERT_FILE,
    SSL_FILETYPE_PEM) != SSL_SUCCESS)
{
    fprintf(stderr, "ERROR: failed to load %s, please check the
        file.\n",
        CERT_FILE);
    return -1;
}

/* Load server key into WOLFSSL_CTX */
if (wolfSSL_CTX_use_PrivateKey_file(ctx, KEY_FILE,
    SSL_FILETYPE_PEM) != SSL_SUCCESS)
{
    fprintf(stderr, "ERROR: failed to load %s, please check the
        file.\n",
        KEY_FILE);
    return -1;
}

/* Initialize the server address struct with zeros */
memset(&servAddr, 0, sizeof(servAddr));

/* Fill in the server address */
servAddr.sin_family = AF_INET;          /* using IPv4 */
servAddr.sin_port = htons(DEFAULT_PORT); /* on DEFAULT_PORT */
servAddr.sin_addr.s_addr = INADDR_ANY; /* from anywhere */

/* Bind the server socket to our port */
if (bind(sockfd, (struct sockaddr *)&servAddr,
    sizeof(servAddr)) == -1)
{
    fprintf(stderr, "ERROR: failed to bind\n");
    return -1;
}

/* Listen for a new connection*/
if (listen(sockfd, 1) == -1)
{
    fprintf(stderr, "ERROR: failed to listen\n");
    return -1;
}
.
.

```

Listing 1: int main() of SSL server, 1* part

This server can communicate with one client at a time so there is an infinite loop where the connection with a client is established. At this point a mainThread is created to handle reading and writing through the secure channel. Whenever the connection falls, the thread ends and the server try to established a new connection. The server ends only when keyboard of the server type 'quit' (this part is in another function).

```
/* Continue to accept clients until shutdown is issued */
while (1)
{
    printf("Waiting for a connection...\n");
    /* Accept client connections */
    if ((connd = accept(sockfd, (struct sockaddr *)&clientAddr,
        &size)) == -1)
    {
        fprintf(stderr, "ERROR: failed to accept the
            connection\n\n");
        return -1;
    }

    /* Create a WOLFSSL object */
    if ((ssl = wolfSSL_new(ctx)) == NULL)
    {
        fprintf(stderr, "ERROR: failed to create WOLFSSL
            object\n");
        return -1;
    }

    /* Attach wolfSSL to the socket */
    wolfSSL_set_fd(ssl, connd);

    /* Establish TLS connection */
    ret = wolfSSL_accept(ssl);
    if (ret != SSL_SUCCESS)
    {
        fprintf(stderr, "wolfSSL_accept error = %d\n",
            wolfSSL_get_error(ssl, ret));
        return -1;
    }
}
```

```

    printf("Client connected successfully\n");
    pthread_t mainThread;
    pthread_create(&mainThread, NULL, ClientHandler, NULL);
    pthread_join(mainThread, NULL);
    printText("Communication is ended!\n", "System");

    if (is_end)
        break;
}
ncurses_end();
printf("Shutdown complete\n");

/* Cleanup after this connection */
wolfSSL_free(ssl); /* Free the wolfSSL object */
close(connd); /* Close the connection to the client */
/* Cleanup and return */
wolfSSL_CTX_free(ctx); /* Free the wolfSSL context object */
wolfSSL_Cleanup(); /* Cleanup the wolfSSL environment */
close(sockfd); /* Close the socket listening for clients */
return 0; /* Return reporting a success */
}

```

Listing 2: int main() of SSL server, 2* part

The previously created thread executes ClientHandler function; even if the connection is established, it shows the 'Client name connected successfully' message on the screen only after reading the username from the secure channel.

After that, two thread are created, one to read messages and one to write messages.

```

void *ClientHandler(void *args)
{
    int ret;
    /****** USERNAME */
    /* Read the client username into our buff array */
    XMEMSET(buff, 0, sizeof(buff));
    ret = wolfSSL_read(ssl, buff, sizeof(buff) - 1);
    ncurses_start();
    clearWin();
    if (ret > 0)
    {
        /* Print to stdout any data the client sends */
    }
}

```

```

        strcpy(username, buff);
        char text[256];
        sprintf(text, "Client %s connected successfully", username);
        printText(text, "System");
        printText("*****\n", "System");
        fflush(stdout);
    }
    else
    {
        printText("ERROR!!", "System");
        close(sockfd); /* Close the connection to the server */
        pthread_exit(NULL); /* End thread execution */
    }
    /****** */
    XMEMSET(buff, 0, sizeof(buff));

    if (pthread_create(&Treader, NULL, readBuffer, NULL))
    {
        fprintf(stderr, "Error creating thread\n");
        fflush(stdout);
        return NULL;
    }

    if (pthread_create(&Twriter, NULL, writeBuffer, NULL))
    {
        fprintf(stderr, "Error creating thread\n");
        fflush(stdout);
        return NULL;
    }

    pthread_join(Treader, NULL);
    pthread_join(Twriter, NULL);
    /* Cleanup after this connection */
    close(connd); /* Close the connection to the client */
    pthread_exit(NULL); /* End thread execution */
}

```

Listing 3: ClientHandler() of SSL server

WriteBuffer is a function executed by a thread for writing in the channel. **read.in()** read the characters typed from the user until a new line and then it sent the message with **wolfSSL_write(ssl,Rbuffer,len)**.

```
void *writeBuffer(void *args)
```

```

{
    int ret;
    while (1)
    {
        read_in();
        len = XSTRLEN(Rbuffer);

        /* Reply back to the client */
        do
        {
            ret = wolfSSL_write(ssl, Rbuffer, len);
            /* TODO: Currently this thread can get stuck infinitely
               if client
            *     disconnects, add timer to abort on a timeout
               eventually,
            *     just an example for now so allow for possible stuck
               condition
            */
            printText(Rbuffer, "Server");

        } while (wolfSSL_want_write(ssl));

        if (XSTRNCMP(Rbuffer, "quit", 4) == 0)
        {
            is_end = 1;
            break;
        }

        if (ret != len)
        {
            printText("ERROR!!", "System");
            break;
        }
    }
    pthread_cancel(Treader);
    return NULL;
}

```

Listing 4: writeBuffer() of SSL server

ReadBuffer function reads the messages from the channel through **wolfSSL_read(ssl, buffReader, sizeof(buffReader)-1)**. If the user on the other side writes 'quit', this function and writeBuffer function end.

```

void *readBuffer(void *args)
{
    int ret;
    while (1)
    {
        /* Read the client data into our buff array */
        XMEMSET(buffReader, 0, sizeof(buffReader));
        ret = wolfSSL_read(ssl, buffReader, sizeof(buffReader) - 1);

        if (ret > 0)
        {
            if (!strcmp(buffReader, "quit"))
            {
                pthread_cancel(Twriter);
                pthread_exit(NULL); /* End thethread execution
                                   */
            }
            printText(buffReader, username);
        }
        else
        {
            printText("ERROR READ!!", "System");
            pthread_cancel(Twriter);
            pthread_exit(NULL); /* End thethread execution      */
        }
    }
}

```

Listing 5: readBuffer() of SSL server

4.5 TLS Client

The TLS client has 3 thread; The main function creates Tclient thread that is used to manage read and write threads.

To execute the TLS client, the program needs the IP address of server. If the user doesn't provide it during the program calls, the program ends.

```

int main(int argc, char **argv)
{
    pthread_t Tclient;
    /* Check for proper calling convention */
    if (argc != 2)
    {

```

```

        printf("usage: %s <IPv4 address>\n", argv[0]);
        return -1;
    }
    ip = argv[1];
    /* create a second thread which executes inc_x(&x) */
    if (pthread_create(&Tclient, NULL, client, NULL))
    {
        fprintf(stderr, "Error creating thread\n");
        fflush(stdout);
        return 1;
    }
    if (pthread_join(Tclient, NULL))
    {
        fprintf(stderr, "Error joining thread\n");
        return 2;
    }
    ncurses_end();
    return 0; /* Return reporting a success          */
}

```

Listing 6: int main() of SSL client

As in the server part, the function below is executed by a thread; it initializes WOLFSSL objects, socket etc. After initializations the function tries to connect to the secure channel, through the IP address inserted at the execution of the client.

```

void *client(void *args)
{
    struct sockaddr_in servAddr;

    printf("Set your username: ");
    refresh();
    if (!scanf("%s", username))
    {
        fprintf(stderr, "ERROR: failed to get message for
            server\n");
        return NULL;
    }
    ncurses_start();
    /* Initialize wolfSSL */
    wolfSSL_Init();

    /* Create a socket that uses an internet IPv4 address,

```

```

    * Sets the socket to be stream based (TCP),
    * 0 means choose the default protocol. */
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
    fprintf(stderr, "ERROR: failed to create the socket\n");
    return NULL;
}

/* Create and initialize WOLFSSL_CTX */
if ((ctx = wolfSSL_CTX_new(wolfTLSv1_3_client_method())) ==
    NULL)
{
    fprintf(stderr, "ERROR: failed to create WOLFSSL_CTX\n");
    return NULL;
}

/* Load client certificates into WOLFSSL_CTX */
if (wolfSSL_CTX_load_verify_locations(ctx, CERT_FILE, NULL) !=
    SSL_SUCCESS)
{
    fprintf(stderr, "ERROR: failed to load %s, please check the
        file.\n",
            CERT_FILE);
    return NULL;
}

/* Initialize the server address struct with zeros */
memset(&servAddr, 0, sizeof(servAddr));

/* Fill in the server address */
servAddr.sin_family = AF_INET;          /* using IPv4 */
servAddr.sin_port = htons(DEFAULT_PORT); /* on DEFAULT_PORT */

/* Get the server IPv4 address from the command line call */
if (inet_pton(AF_INET, ip, &servAddr.sin_addr) != 1)
{
    fprintf(stderr, "ERROR: invalid address\n");
    return NULL;
}

/* Connect to the server */

if (connect(sockfd, (struct sockaddr *)&servAddr,

```



```

        sizeof(servAddr)) == -1)
    {
        printText("ERROR: failed to connect", "System");
        return NULL;
    }
    /*Do something*/

    /* Create a WOLFSSL object */
    if ((ssl = wolfSSL_new(ctx)) == NULL)
    {
        fprintf(stderr, "ERROR: failed to create WOLFSSL object\n");
        return NULL;
    }

    /* Attach wolfSSL to the socket */
    wolfSSL_set_fd(ssl, sockfd);
    /* Connect to wolfSSL on the server side */
    if (wolfSSL_connect(ssl) != SSL_SUCCESS)
    {
        fprintf(stderr, "ERROR: failed to connect to wolfSSL\n");
        return NULL;
    }

    strtok(username, "\n");
    len = strlen(username, sizeof(username));
    /* Send the username to the server */
    if (wolfSSL_write(ssl, username, len) != len)
    {
        fprintf(stderr, "ERROR: failed to write\n");
        return NULL;
    }

    if (pthread_create(&Twriter, NULL, writeBuffer, NULL))
    {
        fprintf(stderr, "Error creating thread\n");
        fflush(stdout);
        return NULL;
    }

    if (pthread_create(&Treader, NULL, readBuffer, NULL))
    {
        fprintf(stderr, "Error creating thread\n");
        fflush(stdout);
    }

```

```

        return NULL;
    }

    pthread_join(Twriter, NULL);
    pthread_cancel(Treader);
    pthread_join(Treader, NULL);

    /* Cleanup and return */
    wolfSSL_free(ssl); /* Free the wolfSSL object */
    wolfSSL_CTX_free(ctx); /* Free the wolfSSL context object */
    wolfSSL_Cleanup(); /* Cleanup the wolfSSL environment */
    close(sockfd); /* Close the connection to the server */
    printText("Communication is ended!\n Press a button!!!",
        "System");
    getch();
    return NULL;
}

```

Listing 7: void *client(void *args) of SSL client

Reading and writing threads are similar to the server part, so I'm not putting them in this text.

4.6 Compile a WolfSSL program

To execute a wolfSSL program you must have installed wolfSSL on your pc, and then you must add **-lwolfssl** on your gcc command.

In this program I used threads and ncurses so I had to add **-pthread** and **-lncurses** flags to gcc command; to optimize the compilation I created a makefile.

4.7 Execute a WolfSSL program

In my project I used a laptop for the client part, and a Raspberry Pi for the server part; they are connected to the same LAN. In the pictures below you can see the GUI of the client and the server part, and the traffic analyze with wireshark. The IP address of server is **192.168.0.53** and the port is **11111**. The IP address of client is **192.168.0.46**

```
pi@raspberrypi:~/Documents/project-wolfssl/code $ ./server-tls-threaded
Waiting for a connection...
```

Figure 8: Execute WolfSSL server

```
luca@luca:~/Documents/latex/miktex/project-wolfssl/code$ ./client-tls 192.168.0.53
Set your username: 
```

Figure 9: Execute WolfSSL client

After the execution of the server, a client can connect to it and the ssl handshake starts.

With the hello packet below, the client provides an ordered list of 27 cipher suites that it will support for encryption. The list is in the order preferred by the client, with highest preference first. This list can be modified by the programmer.

In this case, the client provides a list of optional extension which the server can use to take action or enable new features, for example:

- `key_share`
 - the client sends one or more public keys using an algorithm that it thinks the server will support. This allows the rest of the handshake after the ClientHello message to be encrypted.
- `supported_version`
 - the client indicates its support of TLS 1.3

No.	Time	Source	Destination	Protocol	Length	Info
8134	473.927206832	192.168.0.46	192.168.0.53	TLSv1.3	312	Client Hello
8136	473.954519818	192.168.0.53	192.168.0.46	TLSv1.3	194	Server Hello


```

Frame 8134: 312 bytes on wire (2496 bits), 312 bytes captured (2496 bits) on interface wlp60s0, id 0
Ethernet II, Src: IntelCor_f3:50:e8 (28:c6:3f:f3:50:e8), Dst: Raspberr_95:30:37 (dc:a6:32:95:30:37)
Internet Protocol Version 4, Src: 192.168.0.46, Dst: 192.168.0.53
Transmission Control Protocol, Src Port: 57618, Dst Port: 11111, Seq: 1, Ack: 1, Len: 246
Transport Layer Security
  TLSv1.3 Record Layer: Handshake Protocol: Client Hello
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 241
  Handshake Protocol: Client Hello
    Handshake Type: Client Hello (1)
    Length: 237
    Version: TLS 1.2 (0x0303)
    Random: 415c66d8436758a9fb3260734a5ceb75cff9523aa7a1e07d...
    Session ID Length: 0
    Cipher Suites Length: 54
    Cipher Suites (27 suites)
    Compression Methods Length: 1
    Compression Methods (1 method)
    Extensions Length: 142
  Extension: key_share (len=71)
    Type: key_share (51)
    Length: 71
  Key Share extension
    Client Key Share Length: 69
    Key Share Entry: Group: secp256r1, Key Exchange length: 65
      Group: secp256r1 (23)
      Key Exchange Length: 65
      Key Exchange: 044b90c64339831bce6dc1d731dfda8f59ee518ad9b04ad8...
  Extension: supported_versions (len=3)
    Type: supported_versions (43)
    Length: 3
    Supported Versions length: 2
    Supported Version: TLS 1.3 (0x0304)
  Extension: signature_algorithms (len=32)
    Type: signature_algorithms (13)
    Length: 32
    Signature Hash Algorithms Length: 30
    Signature Hash Algorithms (15 algorithms)
  Extension: supported_groups (len=16)
    Type: supported_groups (10)
    Length: 16
    Supported Groups List Length: 14
    Supported Groups (7 groups)
  Extension: encrypt_then_mac (len=0)
    Type: encrypt_then_mac (22)
    Length: 0

```

Figure 10: Client Hello

In the Server Hello packet, the server has selected cipher suite 0x1301 (TLS_AES_128_GCM_SHA256) from the list of options given by the client. The server also sends a public key using the algorithm of the public key sent by the client. Once this is sent, the rest of the communication will be encrypted.

8134	473.927206832	192.168.0.46	192.168.0.53	TLSv1.3	312 Client Hello
8136	473.954519818	192.168.0.53	192.168.0.46	TLSv1.3	194 Server Hello
<ul style="list-style-type: none"> Frame 8136: 194 bytes on wire (1552 bits), 194 bytes captured (1552 bits) on interface wlp60s0, id 0 Ethernet II, Src: Raspberr_95:30:37 (dc:a6:32:95:30:37), Dst: IntelCor_f3:50:e8 (28:c6:3f:f3:50:e8) Internet Protocol Version 4, Src: 192.168.0.53, Dst: 192.168.0.46 Transmission Control Protocol, Src Port: 11111, Dst Port: 57618, Seq: 1, Ack: 247, Len: 128 Transport Layer Security <ul style="list-style-type: none"> TLSv1.3 Record Layer: Handshake Protocol: Server Hello <ul style="list-style-type: none"> Content Type: Handshake (22) Version: TLS 1.2 (0x0303) Length: 123 Handshake Protocol: Server Hello <ul style="list-style-type: none"> Handshake Type: Server Hello (2) Length: 119 Version: TLS 1.2 (0x0303) Random: 407412cd7050ac796e650fc1d97be4df7d25fa298ee61d48... Session ID Length: 0 Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301) Compression Method: null (0) Extensions Length: 79 Extension: key_share (len=69) <ul style="list-style-type: none"> Type: key_share (51) Length: 69 Key Share extension <ul style="list-style-type: none"> Key Share Entry: Group: secp256r1, Key Exchange length: 65 Group: secp256r1 (23) Key Exchange Length: 65 Key Exchange: 04e1ca5f523c5bd914a3f4898f9e043655e8c901bffcfc376... Extension: supported_versions (len=2) <ul style="list-style-type: none"> Type: supported_versions (43) Length: 2 					
Supported Version: TLS 1.3 (0x0304)					

Figure 11: Server Hello

After the SSL handshake the situation in the server GUI is:

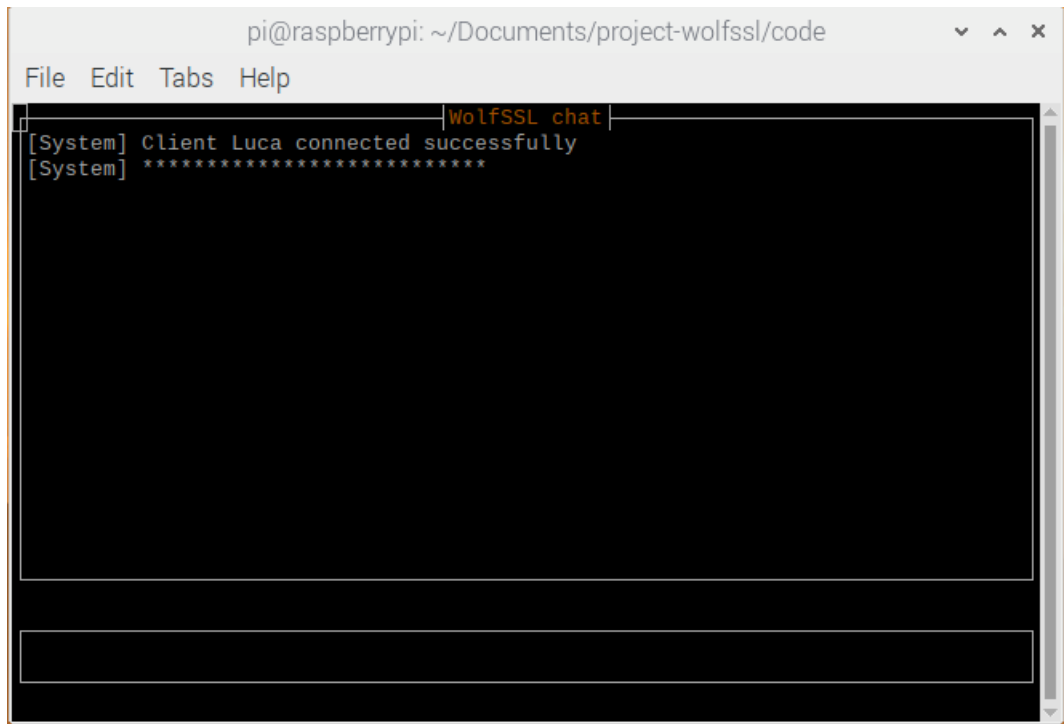


Figure 12: Client connected, server GUI

To test the exact function of the programs, I sent several messages between the laptop and the raspberry:

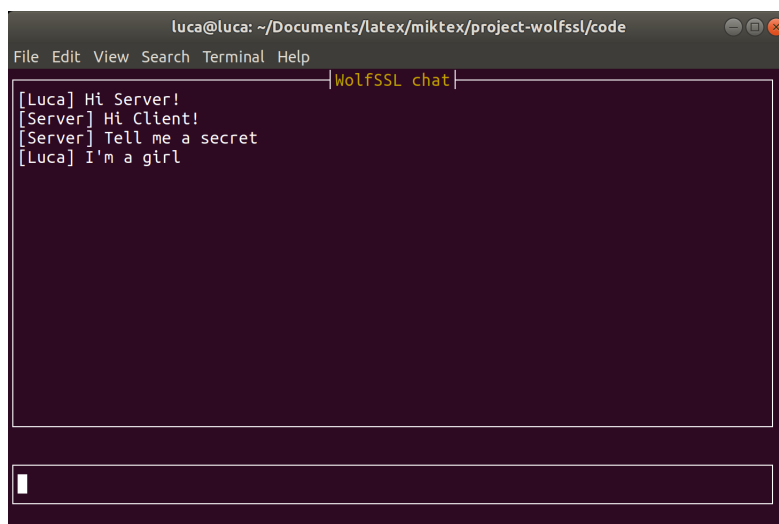


Figure 13: Exchange messages. Client side

```

pi@raspberrypi: ~/Documents/project-wolfssl/code
File Edit Tabs Help

[System] Client Luca connected successfully
[System] *****

[Luca] Hi Server!
[Server] Hi Client!
[Server] Tell me a secret
[Luca] I'm a girl

```

Figure 14: Exchange messages. Server side

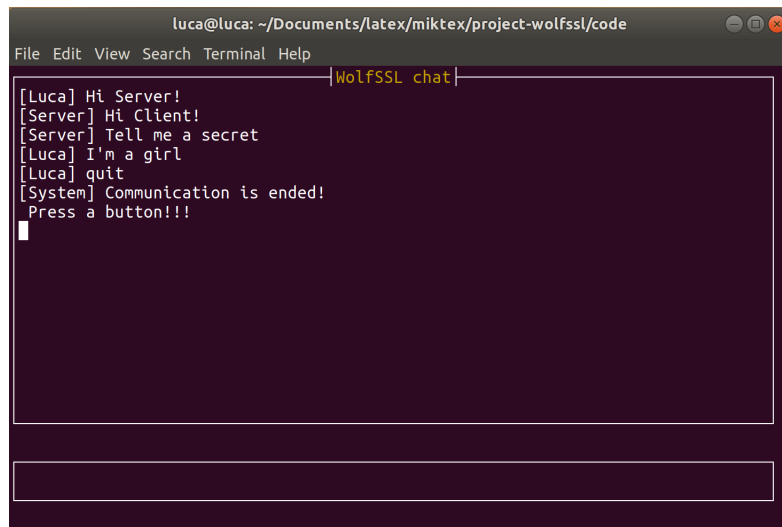
As you can see, the messages are encrypted.

No.	Time	Source	Destination	Protocol	Length	Info
8134	473.927206832	192.168.0.46	192.168.0.53	TLSv1.3	312	Client Hello
8136	473.954519818	192.168.0.53	192.168.0.46	TLSv1.3	194	Server Hello
8138	473.977963222	192.168.0.53	192.168.0.46	TLSv1.3	94	Application Data
8140	473.980204933	192.168.0.53	192.168.0.46	TLSv1.3	1287	Application Data
8142	474.043658750	192.168.0.53	192.168.0.46	TLSv1.3	352	Application Data
8144	474.044790559	192.168.0.53	192.168.0.46	TLSv1.3	124	Application Data
8146	474.045066221	192.168.0.46	192.168.0.53	TLSv1.3	124	Application Data
8148	474.046119492	192.168.0.46	192.168.0.53	TLSv1.3	92	Application Data
8157	478.593312724	192.168.0.46	192.168.0.53	TLSv1.3	98	Application Data
8174	487.004366998	192.168.0.53	192.168.0.46	TLSv1.3	98	Application Data
14963	616.970639892	192.168.0.53	192.168.0.46	TLSv1.3	107	Application Data
15058	629.951492279	192.168.0.46	192.168.0.53	TLSv1.3	98	Application Data
15484	661.111275753	192.168.0.46	192.168.0.53	TLSv1.3	92	Application Data

- Frame 8174: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface wlp60s0, id 0
- Ethernet II, Src: Raspberr_95:30:37 (dc:a6:32:95:30:37), Dst: IntelCor_f3:50:e8 (28:c6:3f:f3:50:e8)
- Internet Protocol Version 4, Src: 192.168.0.53, Dst: 192.168.0.46
- Transmission Control Protocol, Src Port: 11111, Dst Port: 57618, Seq: 1722, Ack: 363, Len: 32
- Transport Layer Security
 - TLSv1.3 Record Layer: Application Data Protocol: tls
 - Opaque Type: Application Data (23)
 - Version: TLS 1.2 (0x0303)
 - Length: 27
 - Encrypted Application Data: 00bac7d5e331e3e2cb5a2f6b5d247743f640de30602ba79f...

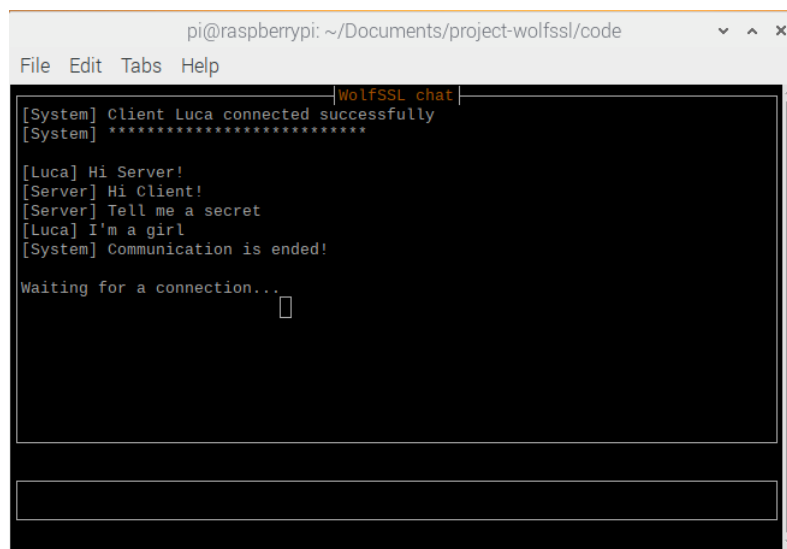
Figure 15: Application data

These are the end of communication screens from the client and from the server:



```
luca@luca: ~/Documents/latex/miktex/project-wolfssl/code
File Edit View Search Terminal Help
wolfSSL chat
[Luca] Hi Server!
[Server] Hi Client!
[Server] Tell me a secret
[Luca] I'm a girl
[Luca] quit
[System] Communication is ended!
Press a button!!!
```

Figure 16: Execute WolfSSL client



```
pi@raspberrypi: ~/Documents/project-wolfssl/code
File Edit Tabs Help
wolfSSL chat
[System] Client Luca connected successfully
[System] *****
[Luca] Hi Server!
[Server] Hi Client!
[Server] Tell me a secret
[Luca] I'm a girl
[System] Communication is ended!
Waiting for a connection...
```

Figure 17: Execute WolfSSL client

5 Differences between WolfSSL and OpenSSL

5.1 Differences

The main differences are:

- Memory Usage

- WolfSSL can be up to 20 times smaller than OpenSSL; The build size is between 20 and 100 KB and the runtime memory usage between 1 and 36 KB. This gives a magior advantage of integrating in smaller embedded devices.
- Hardware Crypto
 - WolfSSL has a partnership with the most MCU manufacturers which allows to be quite early in the market to support hardware acceleration on huge list of platforms.
- Portability
 - WolfSSL is more portable than OpenSSL because is made for real-time, mobile, embedded and enterprise systems.

5.2 Build size and speed test

I installed openssl and wolfSSL in a Raspberry Pi 4 and I could see the difference in occupancy on disk: OpenSSL occupies 412.5 MB while wolfSSL only 47 MB. This difference also affects the compilation and installation time; WolfSSL is compiled and installed in about 3 minutes while openssl takes at least 10 minutes.

Wolfssl is definitely made for embedded systems, but how much can the data forwarding performance differ?

To answer this question I created two program that exchange data between a Raspberry and a laptop, one for wolfSSL and one for openssl. In the specification, one of the two reads a file and sends the read content to the other one. The two programs are identical, the only difference are the function calls. One program has the functions of openssl and the other of wolfssl. For the wolfSSL I used the same certificates and key used in the previous program, while for the openssl I created a root CA certificate, a server key and a certificate signing request with these commands:

```
openssl genrsa -des3 -out CA-key.pem 2048
openssl req -new -key CA-key.pem -x509 -days 1000 -out CA-cert.pem
openssl genrsa -des3 -out server-key.pem 2048
openssl req ?new ?config openssl.cnf ?key server-key.pem ?out
    signingReq.csr
openssl x509 -req -days 365 -in signingReq.csr -CA CA-cert.pem
    -CAkey CA-key.pem -CAcreateserial -out server-cert.pem
```

Listing 8: openssl commands

I created these programs to measure the data forwarding time.
The functions under examination are:

```
void writefile(SSL *ssl, FILE *fp)
{
    ssize_t n;
    char buff[MAX_LINE] = {0};
    clock_t t;
    t = clock();
    while ((n = SSL_read(ssl, buff, sizeof(buff))) > 0)
    {
        total += n;
        if (n == -1)
        {
            perror("Receive File Error");
            exit(1);
        }

        if (fwrite(buff, sizeof(char), n, fp) != n)
        {
            perror("Write File Error");
            exit(1);
        }
        memset(buff, 0, MAX_LINE);
    }
    t = clock() - t;
    double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
    printf("%f seconds to receive data \n", time_taken);
}
```

Listing 9: openssl function

```
void writefile(WOLFSSL *ssl, FILE *fp)
{
    ssize_t n;
    char buff[MAX_LINE] = {0};
    clock_t t;
    t = clock();
    while ((n = wolfSSL_read(ssl, buff, sizeof(buff))) > 0)
    {
        total += n;
        if (n == -1)
        {
```

```

        perror("Receive File Error");
        exit(1);
    }
    if (fwrite(buff, sizeof(char), n, fp) != n)
    {
        perror("Write File Error");
        exit(1);
    }
    memset(buff, 0, MAX_LINE);
}
t = clock() - t;
double time_taken = ((double)t) / CLOCKS_PER_SEC; // in seconds
printf("%f seconds to receive data \n", time_taken);
}

```

Listing 10: wolfSSL function

The rest of the code is on github.

For the following test I created 3 files size about 100 MB, 500 MB and 1 GB with these commands:

```

103.9MB = openssl rand -out sample.txt -base64 $(( 2**30 * 1/14 ))
545.3 MB = openssl rand -out sample.txt -base64 $(( 2**29* 3/4 ))
1 GB = openssl rand -out sample.txt -base64 $(( 2**30 * 3/4 ))

```

Listing 11: openssl commands

This test is made in my gigabit home network. The time is in seconds. (time also includes writing the file)

The cipher suite used is: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

openSSL	time1	time2	time3	average
103.9 MB	3.23	3.29	3.26	3.26
545.3 MB	17.41	17.33	17.38	17.37
1 GB	39.66	36.09	34.01	36.59

wolfSSL	time1	time2	time3	average
103.9 MB	10.48	10.49	10.69	10.55
545.3 MB	55.03	55.13	55.46	55.21
1 GB	109.05	109.48	111.73	110.09

OpenSSL is more performing than wolfssl; the difference of time is relevant. Only by sending large amounts of data you can see the difference between them, but wolfSSL being born for embedded systems works very well and above all it takes up less space.

6 Conclusion

WolfSSL is a beautiful very well done library. They have a forum where thousand of users report their problems and solutions, in which employees too are very active.

A problem of wolfSSL manual is the lack of some details in some sections, such as in the difference with openSSL and its self-promotion orientation.

I think that this thesis can be a good starter guide to get information about wolfSSL since it is difficult to find material from the non official channels.

It is actively being used in a wide range of markets and products including the smart grid, IoT, industrial automation, connected home, M2M, auto industry, games, applications, databases, sensors, VoIP, routers, appliances, cloud services, and more; since wolfSSL is widely employed at a corporate level, there is a great lack of open source projects on the net.

In the future I intend to carry on this research, also focusing on wolfSSL's other products, such as wolfSSH, wolfCrypt, etc.

References

- [1] Official WolfSSL repository
<https://github.com/wolfSSL/wolfssl>
- [2] Official WolfSSL manual
- [3] William Stallings. *Cryptography and Network Security Principles and Practice, Global Edition-Pearson (2017)*
- [4] Rolf Oppliger. *SSL and TLS: Theory and Practice*